A SPARQL 1.1 Query Builder for the Data Analytics of Vanilla RDF Graphs

Sébastien Ferré

A SPARQL 1.1 Query Builder for the Data Analytics of Vanilla RDF Graphs

Sébastien Ferré*
IRISA, Université de Rennes 1
Rennes, France
ferre@irisa.fr

ABSTRACT

As more and more data are available as RDF graphs, the availability of tools for data analytics beyond semantic search becomes a key issue of the Semantic Web. Previous work has focused on adapting OLAP-like approaches and question answering by modelling RDF data cubes on top of RDF graphs. We propose a more direct – and more expressive – approach by guiding users in the incremental building of SPARQL 1.1 queries that combine several computation features (aggregations, expressions, bindings and filters), and by evaluating those queries on unmodified (vanilla) RDF graphs. We rely on the N<A>F design pattern to hide SPARQL behind a natural language interface, and to provide results and suggestions at every step. We have implemented our approach on top of Sparklis, and we report on three experiments to assess its expressivity, usability, and scalability.

KEYWORDS

Data Analytics, RDF Graphs, SPARQL 1.1, OLAP, Expressivity, Query Builder, Natural Language Interface

1 INTRODUCTION

Data analytics is concerned with groups of facts whereas search is concerned with individual facts. Consider for instance the difference between *Which films were directed by Tim Burton?* (search) and *How many films were produced each year in each country?* (data analytics). Data analytics has been well studied in relational databases with data warehousing and OLAP [3], but is still in its infancy in the Semantic Web [4, 10, 11, 13, 18]. Existing work on data analytics for RDF has generally proposed solutions to reproduce the features of OLAP on top of RDF graphs. Typically, data cubes are derived from RDF graphs by specifying what are the observations, the dimensions, and the measures [5]. Traditional OLAP user interfaces can then be used for cube transformations and visualizations. The first limit of this approach is that end-users have no direct access to RDF graphs, and can only explore the cubes that have been defined by some admin-user. This limit is mitigated in [4] by an Analytical Schema (AnS), from which end-users can derive themselves many different cubes. However, there is still the need for an admin-user to define the AnS. The second limit is expressivity. OLAP cubes and their transformations only allow for one level of aggregation, and end-users cannot derive new dimensions or measures from existing one by computation (e.g., population density from total population and area). Semantic Faceted Search (SFS) supports direct exploration of RDF graphs but is even more limited in expressivity w.r.t. data analytics [18].

In contrast, SPARQL 1.1 [19] supports multiple aggregations, nested aggregations with sub-queries, expressions and bindings on numbers, dates, and strings to derive new dimensions/measures [12]. Those features can be mixed with graph patterns with a lot of flexibility. We give concrete use cases that show the need for such a high expressivity. The main reason that prevents end-users to use SPARQL for data analytics is the difficulty to write queries. Question Answering (QA) systems [14], which offer to translate spontaneous user questions to SPARQL, have only recently been extended to RDF data cubes [5], and to statistical question answering thanks to a new task at QALD-6 [20]. However, they exhibit the same limits about direct access and expressivity.

In this paper, we propose an alternative approach that is at the same time more *direct* and more *expressive*. More direct by allowing its application to vanilla RDF graphs, i.e. RDF graphs not customized to data analytics. More expressive by allowing complex and varied combinations of the computation features of SPARQL 1.1 (aggregations, expressions, bindings and filters). Section 2 lists a number of use cases to motivate the need for SPARQL 1.1 expressivity. Section 3 discusses related work. Section 4 presents the principles of our approach, and Section 5 formalizes it as a query builder that guides users in the combination of SPARQL computations. We rely on the N<A>F design pattern [7] to hide SPARQL behind a natural language interface, and to provide results and suggestions at every step. Section 6 presents the implementation of our approach, and Section 7 reports on three experiments on real data, comparing our approach to SPARQL and QA systems, with both IT and non-IT users. Section 8 concludes this paper. Online material[1] related to the paper includes a Web application, the permalinks of about 70 analytical queries, and screencasts.

2 USE CASES

We here present a set of use cases for data analytics on RDF graphs, in the form of questions, in order to clarify our objectives, and to motivate our approach. We first give a few use cases for each kind of SPARQL computation feature (aggregations, and expressions in bindings and filters), and we then give a few use cases combining several computation features. The SPARQL translations of those use cases are given in Figure 1. Use cases are based on a concrete dataset, Mondial [15], to make them more vivid and easier to

*This research was supported by ANR project IDFRAud (ANR-14-CE28-0012-02).

[1] http://bit.ly/sparklis-analytics

```
(A1) SELECT (COUNT(?c) AS ?n) WHERE{?c a :Country; :encompassed :Europe.}
(A2) SELECT ?k (AVG(?p) AS ?ap) (AVG(?a) AS ?aa) WHERE {
       ?c a :Country; :encompassed ?k; :population ?p; :area ?a.
     } GROUP BY ?k
(E) SELECT ?c (?p/?a AS ?d) WHERE{?c a :Country;:population ?p;:area ?a.}
(C) SELECT ?c ?G ?p ?g ?ag WHERE {
     ?c a :Country; :gdpTotal ?G; :population ?p. BIND(?G*1e6/?p AS ?g)
     {SELECT (AVG(?g) AS ?ag) WHERE {
      ?c a :Country;:gdpTotal ?G;:population ?p. BIND(?G*1e6/?p AS ?g)}}
     FILTER(?g > ?ag)}
(N1) SELECT (AVG(?sa) AS ?asa) WHERE {
       {SELECT ?I (SUM(?a) AS ?sa) WHERE {
         ?i a :Island; :belongsToIslands ?I; :area ?a.} GROUP BY ?I } }
(N2) SELECT ?ci (COUNT(?I) AS ?cI) WHERE {
       {SELECT ?I (COUNT(?i) AS ?ci) WHERE {
          ?i a :Island; :belongsToIslands ?I.} GROUP BY ?I }
     } GROUP BY ?ci
```

Figure 1: SPARQL queries for the different use cases

grasp. However, they have nothing specific to that dataset, and could easily be transposed to other datasets. Mondial contains geographical knowledge on countries, cities, continents, bodies of water, etc., and includes the CIA World Factbook. Compared to the BowlognaBench [6], our use cases are at the same time more focused because they consider only computations, and wider because they cover a much larger expressivity range.

Aggregations. A *basic aggregation* is the application of an aggregation operator on a set of entities or values, resulting in a single value. Use case (A1): *How many countries are there in Europe?*. A *simple aggregation* consists in making groups out of a set of values according to one or several criteria, and then applying an aggregation operator on each group of values. An aggregation defines a *cube*, where the grouping criteria are its *dimensions*, and where the aggregated values are its *measures*. A *multiple aggregation* extends simple aggregation by having several aggregated values for each group. Use case (A2): *Give me the average population and the average area of countries, for each continent*. In SPARQL, aggregations rely on the use of aggregation operators in the `SELECT` clause, and on `GROUP BY` clauses. OLAP-like approaches, including existing approaches for RDF, are limited to this category or a subset of it.

Expressions in Bindings and Filters. Direct data analytics sometimes requires other computations than aggregations: e.g., computations on numbers, dates/times, or strings. They correspond to expressions in SPARQL, and can be used either in `BIND` or `FILTER` constructs. Use case (E): *Give me the population density for each country, from population and area*. Semantic Faceted Search systems are limited to comparisons between a variable and a literal.

Complex Combinations. A *complex combination* mixes the previous features in a same question, in any order, and/or several instances of a same feature. SPARQL puts no limit to such combinations. Use case (C) combines a binding, an aggregation, and a filter: *Which countries have a GDP per capita above the average?*. This example requires to compute the GDP per capita as (total GDP $\times 10^6$ / population) for each country (*binding*), to average GDP per capita over all countries (*basic aggregation*), and to select countries whose GDP per capita is higher than the average (*filter*). This use case serves as a running example through Sections 5 and 6.

An interesting kind of complex combination is *nested aggregation*, i.e. an aggregation (basic, simple or multiple) that applies to the result of another aggregation, which can be a nested aggregation itself. Use case (N1): *What is the average area of archipelagos?* (given that the area of each archipelago has first to be summed up from the area of individual islands). Note that the aggregation operator of the nesting aggregation can not only apply to aggregated measures of the nested aggregation but also to dimensions. Use case (N2): *Give me for every number of islands in an archipelago, the number of archipelagos having that number of islands*. In SPARQL, nested aggregations rely on subqueries.

3 RELATED WORK

To the best of our knowledge, our approach is the only one in data analytics of RDF graphs that does not assume the data to be modelled as – or converted to – a datacube at some point. It is also the only one that covers a range of analytical questions beyond the *simple aggregations or OLAP cubes*.

Most existing approaches rely on OLAP. Kämpgen and Harth [13] propose a pipeline to convert statistical linked data for use in OLAP systems. Colazzo et al. [4] propose *analytical schemas* as views over RDF data from which OLAP cubes can be queried in a flexible way. Linked Data Query Wizard [10] is an interface to explore RDF cubes or tables. It supports the filtering of a column (e.g., by a keyword or a numeric value), and simple aggregations. Höffner and Lehmann introduced Question Answering (QA) on RDF Data Cubes (RDCQA) as a subfield of question answering, and designed Task 3 of QALD-6 as a benchmark [11]. Two RDCQA systems have been developed and evaluated in QALD-6: CubeQA by the same authors, and QA^3 [2]. Unlike our approach, OLAP-based approaches (a) require the data to be modelled along the RDF Data Cube Vocabulary [5] so as to identify datasets, dimensions, and measures; (b) are limited to simple aggregations; and (c) require dataset-specific preprocessing and/or templates.

Some approaches allow direct computations on RDF graphs, generally limited to simple filters and aggregations. Neumayr et al. [16] propose ontology-driven RDF analytics to systematically infer statistical facts about classes (e.g., the average age of men, women, people). Sherkhonov et al. [18] propose to extend Semantic Faceted Search (SFS) with filters limited to comparisons between a variable and a number; and with simple aggregations limited to one grouping variable and one triple in the aggregated pattern. Ferré [9] has not those limitations on aggregations but is still limited to simple aggregations.

4 DIRECT AND EXPRESSIVE DATA ANALYTICS OF RDF GRAPHS

Each SPARQL computation feature can be formalized as a function taking a table (of query results) as input, and returning a new table as output. A *binding* extends a table with a new column, whose values are computed from values in other columns according to the binding expression (see use cases (E) and (C)). A *filter* selects a subset of the rows, those rows where the filtering expression evaluates to true (see use case (C)). An *aggregation* groups the rows by some columns, and for each valuation of the grouping columns, applies aggregators (e.g., `COUNT`) to some other columns in order to define one aggregated value for each aggregated column. The result of an aggregation is a table with a column for each grouping column and for each aggregated column, and with a row for each valuation of the grouping columns.

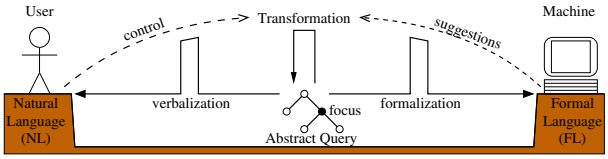

Figure 2: Principle of the N<A>F design pattern [7]

Our ambition is to support all combinations of those computations, only avoiding Cartesian products. The latter restriction is not only for the sake of tractability, but also for the sake of interpretability of results. Combinations naturally include arbitrary chains of computations, e.g., use cases (N1, N2). However, chains are not enough. For example, use case (C) has a filter that compares two columns that are in two different computed tables (the binding and the aggregation). The two tables need to be joined before applying the filter. This situation is not uncommon as it occurs whenever one wants to compare (aggregated) values at different granularity levels (here, GDP at country level vs average GDP at world level).

The problem of extracting the initial tables from an RDF graph amounts to the problem of composing a SPARQL graph pattern. The table of a graph pattern P is the result of the SPARQL query `SELECT * WHERE {`P`}`. Semantic faceted search enables non-IT users to interactively build such patterns over RDF graphs. It makes it unnecessary to rely on experts to prepare tables of data, and it therefore allows for direct analysis of RDF data. The two most advanced systems seem to be Sparklis [9] and SemFacet [1]. While the latter supports OWL2 reasoning, the former supports a much larger set of patterns: i.e., projection of several variables, cycles (not only tree patterns), negation (`MINUS`), and optionals (`OPTIONAL`). In particular, the capability to project several variables, i.e. to produce tables and not only sets, is essential to data analytics. Note that, as our approach only assumes an initial table to start with, it is also applicable to any context where such tables are available (e.g., relational databases, spreadsheets, OLAP cubes).

Compared to relying on RDF data cubes [5], direct data analytics on vanilla RDF graphs offers the following advantages. Arbitrary graph patterns can be built to specify the observations, dimensions, and measures of cubes, without the need for RDF data cube modelling or pre-processing. For example, to answer the question *What is the total area of lakes per country?*, other approaches would need a data cube where countries and lakes are two dimensions, and where the area is a measure. However, a better and more natural RDF modelling is to have a property from lakes to their countries, and another property from lakes to their area. In fact, we think that RDF data cube models have more to do with query patterns than with data models. Indeed, a data cube can be seen as a SPARQL query template where it remains to choose the projected variables (dimensions), and the aggregation operators.

5 BUILDING ANALYTICAL QUERIES

We address the problem of composing analytical queries by guiding users in the incremental building of SPARQL computation queries, and by hiding SPARQL behind a Natural Language Interface (NLI). We formalize our approach using the N<A>F design pattern [7]. Its purpose is to bridge the gap between a natural language (NL) and a formal language (FL, here SPARQL). Figure 2 summarizes N<A>F. The user stands on the NL side, and does not understand the FL. The machine, here a SPARQL endpoint, stands on the FL side, and does not understand the NL. The central element of the bridge is an *abstract query*. In short, it is a representation of the user's question in a language that is intermediate between NL and FL. That language is designed to make translations from abstract queries to both NL (*verbalization*) and FL (*formalization*) as simple as possible. A sub-query of the abstract query is distinguished as the *query focus*, and the remainder is called *query context*. N<A>F includes the query builder approach, where the structure that is incrementally built is precisely the abstract query. The abstract query is initially the simplest query, and it is incrementally built by applying transformations. A *transformation* may insert or delete a query element, or move the focus. N<A>F also includes the faceted search approach, where transformations are suggested by the machine based on query semantics and actual data, and controlled by users. Like queries, transformations are verbalized in NL. This section successively defines abstract queries, their formalization in SPARQL, their verbalization in English, their transformations, and finally the type-based suggestion of transformations. All definitions are illustrated on complex use case (C).

Table 1: Table queries, along with their columns $C(T)$, their dimensions $D(T)$, and their well-formedness conditions

table query T	$C(T)$	$D(T)$	well-formedness conditions
$Pattern(X, P)$	X	X	$X \subseteq Vars(P)$
$Bind(T_1, x, E)$	$C(T_1) \cup \{x\}$	$D(T_1)$	$X_E \subseteq C(T_1)$, x fresh
$Filter(T_1, E)$	$C(T_1)$	$D(T_1)$	$X_E \subseteq C(T_1)$, E Boolean
$Aggreg(T_1, X, Y, G, Z)$	$X \cup Z$	X	$X \uplus Y \subseteq C(T_1)$, Z fresh
$T_1 \triangleright T_2$	$C(T_1) \cup C(T_2)$	$D(T_1)$	$D(T_2) \subseteq D(T_1)$,

5.1 Abstract Queries

We call our abstract queries *table queries* (T) because their evaluation returns tables. Table queries are recursively composed of other table queries, and are also composed of *expressions*.

Definition 1 (Expressions). *An* expression *is one of:*

- *n, where n is an RDF node (URI or literal),*
- *?x, where x is a table column,*
- *E_1 op E_2, where op is a binary SPARQL operator (e.g., +) applied to two sub-expressions,*
- *$f(E_1, \ldots, E_n)$, where f is a SPARQL function (e.g., `str`) applied to a tuple of sub-expressions as arguments,*
- *??, representing an undefined expression (yet to be built).*

We note X_E the set of columns occuring in expression E.

Definition 2 (Table queries). Table queries *take one of the forms given in Table 1 (left column). A Pattern-query is a primitive table query, $\triangleright$ is a binary operator, and Bind, Filter, Aggreg are unary operators. Those unary operators represent the computation of a table as a function of another table, as informally presented in Section 4.*

Table 2: Translation of table queries to SPARQL patterns

T	$Pattern_T$
$Pattern(X, P)$	P
$Bind(T_1, x, E)$	$Pattern_{T_1}$ `BIND (`$expr_E$ `AS ?`x`)`
$Filter(T_1, E)$	$Pattern_{T_1}$ `FILTER (`$expr_E$`)`
$Aggreg(T_1, X, Y, G, Z)$	`{SELECT ...?`x_i `...(`g_j`(?`y_j`) AS ?`z_j`)...` `WHERE {` $Pattern_{T_1}$ `}` `GROUP BY ...?`x_i `...}`
$T_1 \triangleright T_2$	$Pattern_{T_1}$ $Pattern_{T_2}$

Table 1 also gives for each form the resulting set of columns $C(T)$, the set of dimensions *$D(T) \subseteq C(T)$, and well-formedness conditions. $D(T)$ plays the role of a key for the resulting table T (i.e., at most one row for each valuation of $D(T)$).*

$Pattern(X, P)$ represents an initial table from the SPARQL pattern P, where the set of columns X is a subset of the variables in P. $Bind(T_1, x, E)$ extends T_1 with a new column x computed by expression E over T_1's columns. $Filter(T_1, E)$ filters the rows of table T_1 based on the Boolean expression E. $Aggreg(T_1, X, Y, G, Z)$ defines an aggregated view over T_1, grouping by columns in X, and aggregating each column $y_j \in Y$ with aggregator $g_j \in G$ into aggregated column $z_j \in Z$. Finally, $T_1 \triangleright T_2$ is a restricted form of join where $D(T_2) \subseteq D(T_1)$ (the key of T_2 is a subset of the key of T_1), so that each T_1-row has at most one corresponding T_2-row.

Example. In order to illustrate our language of abstract queries, and show its expressive power, we show below the table query T for the complex use case (C) of Section 2. For readability, we decompose the table query by naming each sub-query (T_0, T_1, T_2). This also avoids the duplication of T_1, which is used twice.

T = Filter($T_1 \triangleright T_2$, (?g > ?ag))
  where T_2 = Aggreg(T_1, (), (g), (AVG), (ag))
  and T_1 = Bind(T_0, g, ((?G * 1e6) / ?p))
  and T_0 = Pattern({c,G,p},
    `?c a :Country ; :gdpTotal ?G ; :population ?p .`)

T_0 is the initial table based on a SPARQL pattern relating countries to their total GDP (in M$), and their population. T_1 extends T_0 with a binding computing the GDP per capita (column g). T_2 performs an aggregation over T_1 to produce a one-row one-column table containing the average GDP per capita (column ag). Finally, T filters the join $T_1 \triangleright T_2$ by comparing columns g from T_1 and column ag from T_2. The join is here necessary in order to make the two columns accessible to the filter expression from a same table. For final table T, the set of columns is $C(T) = \{c, G, p, g, ag\}$, and the set of dimensions is $D(T) = \{c, G, p\}$ because g and ag are introduced by a binding and an aggregation, respectively.

5.2 Formalization in SPARQL

We here provide a formalization and semantics for abstract queries by translating them to SPARQL. Although the translation of each table query and expression is relatively straightforward (see Table 2 for tables), it is complicated by the fact that the table query to be translated depends on the current query focus. Indeed, the idea is to give access not only to the final table, but also to the initial table and every intermediate table (each T_i in the above example). The current focus can be used to select the sub-query to translate. For example, in use case (N1), the user can choose between: (1) islands and their archipelagos and area, (2) archipelagos and their total area, and (3) the average area of archipelagos.

However, it is also desirable to show as much information as possible, by joining the table at focus with other compatible tables. For example, in use case (C), when the focus is on the initial table about countries, it is valuable to include the column on "GDP per capita" from the binding (T_1), and also the "average GDP per capita" from the aggregation (T_2). In particular, it makes it possible to compare side-by-side for each country its GDP per capita to the average. To formalize this intuition, we define the *saturation* of a table sub-query.

Definition 3. *Let T be a query table, and T_i be a sub-query of T (or T itself). The* saturation *of T_i is $T_i^{\triangleright} = T_i \triangleright \{T_j \subseteq T \mid T_j \neq T_i, D(T_j) \subseteq D(T_i)\}$, i.e. the join of T_i with all other sub-queries of T whose dimensions are a subset of the dimensions of T_i.*

In the above example query T, the saturation of T_0 is $T_0^{\triangleright} = T_0 \triangleright T_1 \triangleright T_2 \triangleright T$, which simplifies to $T_0^{\triangleright} = T$ thanks to the following rules: $T \triangleright Bind(T, x, E) = Bind(T, x, E)$, and $T \triangleright Filter(T, E) = Filter(T, E)$. The saturation of T_1 is the same. The saturation of T_2 is $T_2^{\triangleright} = T_2$ because $D(T_2) = \emptyset$, unlike other sub-queries.

Definition 4. *The SPARQL translation of a table query T is*

`SELECT * WHERE {` $Pattern_{T_{focus}^{\triangleright}}$ `}`,

where T_{focus} is the sub-query that has focus, $T_{focus}^{\triangleright}$ is its saturation, and $Pattern_{T_{focus}^{\triangleright}}$ is defined recursively on table queries by Table 2.

In Table 2, $expr_E$ represents the translation of our abstract expression E to SPARQL expressions. The translation of expressions is trivial but the focus and undefined expressions (??) have to be taken into account again. First, when an expression under focus contains the undefined expression ??, the encompassing `BIND` or `FILTER` is removed from the translation because the expression cannot yet be computed. Second, when the focus is on a sub-expression, the rest of the expression is ignored in the translation in order to give access to the values of that sub-expression. For example, if the focus is on `(?G * 1e6)` in use case (C), then the rest of the expression (`/ ?p`) is ignored, and column g contains values of the "total GDP in dollars" instead of values of the "GDP per capita".

Example. The formalization of use case (C) when the focus is on one of T_0, T_1, T, and not on a sub-expression is the following.

```
SELECT ?c ?G ?p ?g ?ag WHERE {
  ?c a :Country ; :gdpTotal ?G ; :population ?p .
  BIND (?G*1e6/?p AS ?g)
  { SELECT (AVG(?g) AS ?ag) WHERE {
      ?c a :Country ; :gdpTotal ?G ; :population ?p .
      BIND(?G*1e6/?p AS ?g) } }
  FILTER (?g > ?ag) }
```

Note the duplication of the initial pattern and the binding, coming from the two occurences of T_1 in table query T. When the focus is on T_2, the formalization is only the aggregation query, i.e. a subquery of the previous one.

```
SELECT (AVG(?g) AS ?ag) WHERE {
  ?c a :Country ; :gdpTotal ?G ; :population ?p .
  BIND(?G*1e6/?p AS ?g) }
```

5.3 Verbalization in English

Before defining the verbalization of expressions and table queries, we have to define the verbalization of column names. We assume that a noun is associated to each column of a *Pattern*-query: ex., 'country' for c, and 'total GDP' for G in the running example. Columns introduced by bindings can be named by users: ex., 'GDP per capita' for g. Otherwise, the verbalization of the binding expression is used instead. Finally, the verbalization of each aggregated column z is the verbalization of $g(y)$ (e.g., 'the number of y', 'the average y').

The verbalization of expressions results from the nesting of the verbalization of its functions and operators It mixes mathematical notations and text depending on which is the clearer: 'E_1 + E_2' is clear to everybody and less verbose than 'the addition of E_1 and E_2', while 'E_1 or E_2' is less obscure than 'E_1 || E_2' for non-IT people. The verbalization of table queries is a matter of templates (optional parts are between [ ]), given the verbalization of columns, aggregated columns, and expressions.

- $Bind(T_1, x, E)$: 'give me [x =] E'
- $Filter(T_1, E)$: 'where E'
- $Aggreg(T_1, X, Y, G, Z)$: '[for ... each x_i ... ,] give me ... $g_j(y_j)$ ... '

All sub-queries in a table query are verbalized once – hence avoiding duplications for sub-queries that have several occurences (e.g., T_1 in use case (C)) – and coordinated with 'and'. Sub-queries are put in dependency ordering, starting with the initial table. In agreement with formalization, if the focus is on sub-query T_i then the sub-queries that do not appear in the saturation $T_i^{\triangleright}$ are dimmed in the display to show that they are inactive at the current focus. Here is the verbalization of use case (C), whose first line is the verbalization of the initial pattern, as done in Sparklis [9].

```
give me a country that has a total GDP, and that has a population
and give me GDP per capita = the total GDP * 1e6 / the population
and give me the average GDP per capita
and where the GDP per capita > the average GDP per capita
```

5.4 Query Transformations

A query transformation applies changes to an abstract query at and around its focus. This notion of focus is essential to target the part of the query to be changed, and to simplify the definition of the transformations. Each of our transformations is represented in the following way:

$$focus@context \xrightarrow{transf} focus'@context'$$

where a pair *focus@context* exhibits the split of an abstract query between the sub-query or sub-expression at focus, and its context, i.e. the surrounding query. For example, $E@Filter(T, \_)$ denotes the query $Filter(T, E)$ with focus on E.

The simplest way to have a complete set of transformations would be to have a focus for each table sub-query and sub-expression, and to have an insertion transformation for each form of table queries and expressions: i.e., bindings, filters, aggregations, joins, nodes, columns, operators, and functions. However, this would not be satisfying for several reasons. First, it would force users to think in terms of table queries (e.g., bindings, filters), whereas it is more natural to think in terms of entities, values, and function/operator applications. Therefore the focus should be on a column or an expression instead. Second, it would force to compose expressions top-down, which would require anticipation of the whole expression. It should be possible to compose expressions in a mixture of top-down and bottom-up, and even to insert functions/operators in the middle of an expression.

Table 3: Query transformations (see Definition 6)

(0) $x@T \xrightarrow{f[1]} f(?x, ??)@Bind(T^{\triangleright}, x, \_) \xrightarrow{??}$
(1) $E@E' \xrightarrow{f[1]} f(E, ??)@E' \xrightarrow{??}$
(2) $??@E' \xrightarrow{n} n@E' \xrightarrow{??}$
(3) $??@E' \xrightarrow{x} ?x@E' \xrightarrow{??}$
(4) $??@E' \xrightarrow{f} f(??, ??)@E' \xrightarrow{??}$
(5) $E@Bind(T, x, \_) \xrightarrow{type(E)=bool} E@Filter(T, \_)$
(6) $E@Filter(T, \_) \xrightarrow{type(E)\neq bool} E@Bind(T, x^*, \_)$
(7) $y@T \xrightarrow{g} z^*@Aggreg(T^{\triangleright}, (), (y), (g), (z^*))$
(8) $c@Aggreg(T, X, Y, G, Z) \xrightarrow{group\ by\ x} x@Aggreg(T, (Xx), Y, G, Z)$
(9) $c@Aggreg(T, X, Y, G, Z) \xrightarrow{g(y)} z^*@Aggreg(T, X, (Yy), (Gg), (Zz^*))$

Definition 5 (query focus). *There are two kinds of* query focus*:*

- *$x@T$ sets the focus on one column $x \in C(T)$ of table query T;*
- *$E@E'$ sets the focus on an expression E, whose context is E'. Context E' can be refined into $Bind(T, x, \_)$ or $Filter(T, \_)$.*

Definition 6 (query transformations). *The* query transformations *of table queries and expressions are listed in Table 3. The starred columns represent the introduction of fresh columns. The trailing ($\xrightarrow{??}$) in Transf. (0-4) represents an automatic move of the focus to the next undefined expression if there is any, or to the whole expression otherwise, in order to save manual focus moves.*

Transf. (0) introduces a binding by applying function f to column x, as the 1st argument of f. Transf. (1) inserts a function call on any expression focus. For brevity, those two transformations are only given for binary functions and 1st arguments, but they are also defined for operators, other function arities, and other argument positions (e.g., `*[1]`, `concat[2]`). Transf. (2,3,4) replace an undefined expression with one of: a node n, column x (when $x \in C(T)$, where T is the focus table), or a call to function f with arguments waiting to be defined. Transf. (5,6) represent automatic switches between bindings and filters based on the type of the expression being Boolean or not. Transf. (7) introduces an aggregation by applying aggregator g to column y. Transf. (8,9) respectively add a grouping by column x (when $x \in C(T) \setminus X \setminus Y$) and an aggregated column $g(y)$ (when $y \in C(T) \setminus X$) to an aggregation. Note the use of saturation when introducing bindings (Transf. (0)) and aggregations (Transf. (7)) in order to give access to as many columns as possible for defining the binding or aggregation. In fact, there is no transformation to insert a join, and all joins are automatically inserted in this way. For brevity, we have omitted additional transformations for removing query elements. They are not necessary in theory but they are very useful in practice for undoing insertions in a different

order to their insertion. In addition to those transformations, the focus can be moved freely.

Only transformations that are applicable to the current focus, and that respect type constraints are suggested (see Section 5.5). The suggested operators and functions in Transf. (0,1,4), and the aggregators in Transf. (7,9) are selected from the list of SPARQL-supported operators, functions, and aggregators, according to type constraints. The suggested columns in Transf. (3) are the columns $C(T^{\triangleright}_{focus})$ of the saturation of the focus table. The suggested columns in Transf. (8,9) are taken among the columns of the aggregated table T, according to the well-formed conditions of *Aggreg*-tables (Table 1). Finally, the nodes in Transf. (2) are obtained by asking the user to fill an input field, and by checking the input according to type constraints.

Example. The sequence of transformations that leads, in use case (C), from the initial table (T_0) to the complete query (T) is the following, where each transformation, and each focus move (e.g., $G@T_0$), is followed by the table query that has been created or modified. In the latter, the query focus is underlined. We recall that transformations are not written by the user but selected (by click) in a suggestion list.

step	transf.	created or modified table query
1	G@T_0	$T_0 \leftarrow$ Pattern({c,G,p},…)
2	*[1]	$T_1 \leftarrow$ Binding(T_0, g, (?G * ??))
3	1e6	$T_1 \leftarrow$ Binding(T_0, g, (?G * 1e6))
4	/[1]	$T_1 \leftarrow$ Binding(T_0, g, ((?G * 1e6) / ??))
5	p	$T_1 \leftarrow$ Binding(T_0, g, ((?G * 1e6) / ?p))
6	AVG	$T_2 \leftarrow$ Aggreg(T_1, (), (g), (AVG), (ag))
7	g@T_1	$T_1 \leftarrow$ Binding(T_0, g, ((?G * 1e6) / ?p))
8	>[1]	$T \leftarrow$ Filter($T_1 \triangleright T_2$, (?g > ??))
9	ag	$T \leftarrow$ Filter($T_1 \triangleright T_2$, (?g > ?ag))

The above sequence of transformations can be informally presented in a narrative way (imagine the user talking to herself): from the initial table, *"select the total GDP... multiply it by...* 10^6 *... and divide the result by... the population (the result is the GDP per capita)... get the average of that... select the GDP per capita... it should be greater than... the average GDP per capita"*. It can be observed that 9 steps are sufficient to build the whole table query although it contains 5 sub-queries, and 8 sub-expressions. It can also be observed that every applied transformation is either referring to a column (2 focus changes, 2 column insertions), or inserting a computing element (3 operators, 1 number, 1 aggregator). Therefore, users can think in terms of which computations should be applied to which columns, and table queries are implicitly created/modified as needed. Note that, despite T_1 having two occurences in T, it needs only be built once thanks to the automatic joins.

5.5 Type-Based Suggestions

Among the possible transformations presented in Section 5.4, only those that lead to well-typed expressions and aggregations should be suggested to users. For example, when the focus is ??@(?G * _), i.e. on the right-hand side of a product, only *numeric* literals, columns and functions can be inserted. We have implemented a *type inference* mechanism that determines two type constraints: the *focus type* and the *context type*. The *focus type* comes from the analysis of the query focus, and possibly the focus values. If the query focus is an expression, then the focus type is infered from the type signatures of its functions and operators. Similarly for aggregations. If the query focus is a pattern column, the focus type is infered from the datatype of its values in the SPARQL results. If the query focus is an undefined expression, which happens when inserting an n-ary function, the focus type is undefined. The *context type* comes from the analysis of the query context. If the query context is a function/operator argument, then the context type is the type of that argument, otherwise it is undefined. By definition of the transformations, at least one of the two constraints is defined. From there, the suggested aggregators, functions, and operators are simply those that can be inserted at the focus while satisfying the defined type constraints. In fact, there is a distinct transformation for each function argument (Section 5.4), and only function arguments compatible with focus type are suggested. The type constraints are also used to check user inputs of literals.

In practice, a few additional complications must be dealt with for the sake of robustness w.r.t. real datasets. First, the focus values may have several incompatible types (e.g., URIs and strings). In this case, the focus type is a set of types, and if the inserted function is only compatible with one of them, the resulting expression will only be partially defined (there will be blank cells in the results). Second, the focus values may be improperly typed, thus requiring to explicitly apply conversion functions. The common case is numerical literals having no datatype (e.g., QALD-6 datacubes) or non-standard datatypes (e.g., DBpedia `USdollar`). We parse literals independently of their datatype to recognize numbers, and implicitly apply conversion functions in the SPARQL formalization as necessary.

6 IMPLEMENTATION

We have fully implemented the proposed approach on top of Sparklis [9], a semantic search tool that is also based on the N<A>F design pattern. It allows end-users to build SPARQL graph patterns that arbitrarily combine basic graph patterns, `UNION`, `OPTIONAL`, and `MINUS`. It also supports `ORDER BY` clauses. It therefore provides everything needed for the building of *Pattern*-table queries, and our proposed approach fits well with Sparklis.

Our implementation, called Sparklis-Analytics, incorporates Sparklis, and adds 3500 lines of code to the existing 5000. The N<A>F design pattern helped us to integrate table queries and expressions in a modular way. First, new data structures were defined beside patterns for table queries and expressions. Second, the algorithms for the formalization in SPARQL and the verbalization in English were extended to those new data structures. Third, the set of suggestions was augmented with our computation-oriented transformations. Finally, type-checking mechanisms were added for the filtering of suggested transformations.

Availability. Sparklis-Analytics is available on-line[2] as a client-side application that works on top of SPARQL endpoints. A few endpoints are proposed (e.g., DBpedia, Mondial, Nobel Prizes) but any endpoint can be explored by simply entering its URL. A few configuration options allow to adapt to different endpoints (e.g., `GET`/`POST` method, sending with credentials), and to specify labelling properties. It also includes the YASGUI editor [17] to let

[2] `http://bit.ly/sparklis-analytics`.

advanced users access and modify the SPARQL translation of the query. The application page links to a list of clickable examples including the use cases of Section 2 (and more), and the 50 test questions from the QALD-6 challenge. A number of examples have a screencast on YouTube to show their incremental building.

Example. Figure 3 shows a screenshot of Sparklis-Analytics, taken at the last step of use case (C), just before inserting `'the average GDP per capita'` (column *ag*) in the filtering expression[3]. The user interface has three main parts, from top to bottom: the query, the suggested transformations organized in three lists, and the table of results. In the query, the focus is on the right-hand side of operator >, and the left-hand side is dimmed because it is not under focus. The middle list suggests column names whose type is compatible, and entry fields for compatible types (here, numeric types). The right list suggests functions and operators whose type is compatible (here, operators returning a number). The left list is not relevant for computations, it is used for building graph patterns (suggesting classes and properties). In the table of results, we can see the active column names, and the first row. Both the computed `'GDP per capita'` and `'average GDP per capita'` are active, which allows to compare them side by side. The last column is added to show the Boolean value of the comparison expression under building but it contains only blank cells because the focus is undefined at this stage.

Expressivity and limits. Our approach and its implementation covers almost all SPARQL 1.1 computations. The non-covered features are: (a) a few technical functions like `SHA256`, (b) the customization of the separator in the `GROUP_CONCAT` aggregator, and (c) the application of `COUNT` to a row of variables (e.g., `COUNT(*)`). The limits therefore do not lie in the computation features but in their combinations. A hard limit is that a query cannot contain several disconnected patterns because our join ($T_1 \triangleright T_2$) is the only way to combine several patterns, and its restriction ($D(T_2) \subseteq D(T_1)$) implies that the two patterns share variables, and hence are connected. An example of question that cannot be answered in our approach is "Are there more cities than rivers?", because cities and rivers are not connected. However, it is possible to answer the question "Are there more cities than rivers in France?", because cities and rivers are now connected through entity France. A soft limit is that some questions, although answerable in our approach, are awkward to build. The typical example is the computation of proportions, and more generally, the combination of aggregations on different filterings of a table query: e.g., "What proportion of cities in China have more than 1 million inhabitants?".

Scalability. The addition of computation features to Sparklis has not much impact on scalability, apart from the SPARQL evaluation of aggregations that are intrinsically costly, in particular in complex combinations. Bindings and filters do not impact substantially efficiency as they are evaluated row-wise. The cost of the translations from abstract queries to SPARQL and English is negligible compared to the cost of evaluating SPARQL queries by the endpoint. The type-based choice of suggestions is cheap because it can be computed on the client side without any request to the SPARQL endpoint. Our experiments in real settings (Section 7) confirm those theoretical observations.

[3]We warmly invite the reader to look at the full screencast (2'54") of example (C) at `https://youtu.be/MPiYMwxasFo`.

7 EXPERIMENTS

We here report on three experiments in order to evaluate our approach. The first is a user study that compares the usability of our approach to the writing of SPARQL queries. The second reports on our participation to the QALD-6 challenge, and compares our results to question answering systems. The third evaluates the adequecy of our approach w.r.t. real needs at Persée, a dataset used by researchers in social sciences.

7.1 Comparison with Writing SPARQL Queries

Methodology. The objective of this user study is to evaluate the usability of our approach compared to writing SPARQL queries directly. The subjects were 24 post-graduate students working in pairs. They had all recently attended a Semantic Web course with about 10h of teaching and practice of SPARQL. The task given to subjects was to answer two series of 10 questions on the Mondial dataset, covering all categories presented in Section 2. The questions in the two series involve exactly the same kinds of computations in the same order, they only differ by the initial pattern: e.g. *Give the average population of European countries* vs *Give the average area of Mediterranean islands*. The subjects had to use two different systems, one for each series of questions: (a) YASGUI [17], a SPARQL query editor improved with syntax highlighting, and (b) our tool Sparklis-Analytics. The subjects had only a 1h presentation of our tool before, and no practice of it. Each subject was randomly assigned an order (YASGUI first vs Sparklis-Analytics first) to avoid bias. For each system/series couple, they were given 10min for setup, and 45min for question answering. To help the writing of SPARQL queries, they were given the list of classes and properties used in Mondial. Finally, the subjects filled a questionnaire at the end to report their feelings and comments.

Questions. Question 1 is a simple pattern. Question 2 is a binding, and Question 3 is a filtering. Questions 4-7 are aggregations with different numbers of groupings. Question 8-9 are complex combinations of an aggregation and a binding or filtering. Question 10 is a nested aggregation.

Objective results. With YASGUI, the subjects managed to produce answers for 1-3 questions, 1.67 on average. The rate of correct answers was 71%. The best subject answered to 3 questions, all correctly, and with an average of 15min per question. In comparison, with Sparklis-Analytics, the subjects managed to produce answers for 3-10 questions, 6.17 on average. This is 3.7 more questions, and the weakest result for our tool is equal to the strongest result for YASGUI. The rate of correct answers is also higher at 85%. Most errors (8/13) were done by only 2 subjects out of 12. The most common error is the omission of groupings in aggregations, suggesting to make them more visible in the user interface. The best subject answered to 10 questions, 8 of which were correct, and with an average of 4.5min per question. The time spent per question is generally higher for the first 2 questions, and generally stays under 10min for the other questions, although they are more complex, and can be as low as 2-3min. Those results demonstrate that, even without practice of our tool, subjects quickly learned to

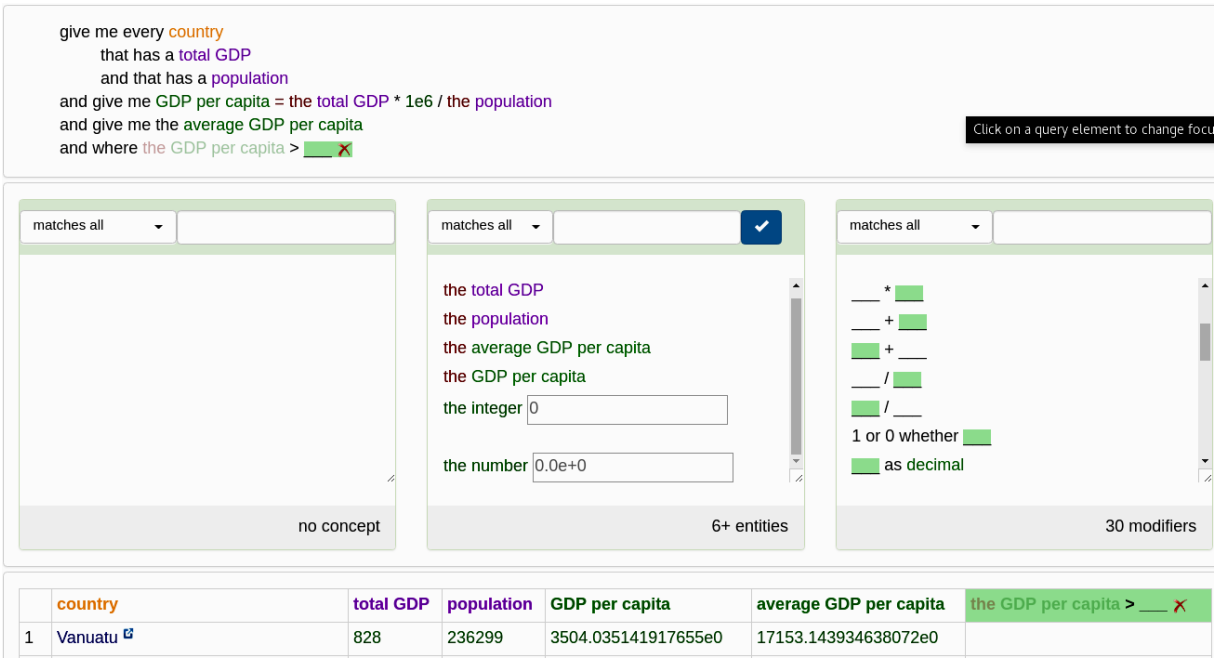

Figure 3: Screenshot of Sparklis-Analytics at the last step of use case (C)

use it, and are much more effective with it. The weakest subject on Sparklis-Analytics is still as good as the strongest subject on YASGUI. Moreover, all subjects had better results with the former than with the latter.

Subjective results. As a first question, we asked subjects whether they (1) clearly prefer YASGUI, (2) would rather use YASGUI, (3) have no preference, (4) would rather use Sparklis-Analytics, or (5) clearly prefer Sparklis-Analytics. Over the 12 subjects, 9 *clearly prefer* Sparklis-Analytics and 3 *would rather use* Sparklis-Analytics, hence unanimous preference for our tool. As a second question, we asked them to evaluate the two systems on a 0-10 scale. YASGUI got marks between 2 and 8, 4.8 on average. Sparklis-Analytics got marks between 7 and 10, 8.3 on average (6/12 subjects gave 9-10 marks). From two other questions, we know that (a) 11/12 subjects think that *knowledge about Semantic Web technologies is not necessary to use* Sparklis-Analytics, and (b) 10/12 subjects said that *non-IT people would quickly learn how to use* Sparklis-Analytics. When asking them what they preferred in our tool compared to writing SPARQL, most said that "it is easy-to-use and intuitive", and that "it is not necessary to make searches out of the application to find classes and properties".

7.2 QALD-6 Datacube Challenge

In this experiment, we evaluated our approach on a large and real dataset, and on a challenging set of questions. The QALD-6 challenge (Question Answering over Linked Data) introduced a new task (Task 3) on "Statistical question answering over RDF datacubes" [20]. The dataset contains about 4 million transactions on government spendings all over the world, organized into 50 datacubes. Note that organisation into datacubes is not required by our approach, unlike in other approaches. There are about 16M triples in total. For the evaluation, we used our tool with the following configuration: `http://cubeqa.aksw.org/sparql` as endpoint, and `rdfs:label` as labelling property for entities, classes and properties.

Table 4: Performance on QALD-6 statistical questions by two different users

user	answered questions	correct answers	min/median/max time
expert	49/50 = 0.98	47/50 = 0.94	31"/1'30"/6'20"
beginner	44/50 = 0.88	38/50 = 0.76	1'/4'/10'

Questions. All 100 training questions and 50 test questions are *basic or simple aggregations*, except for two questions (training Q23 and test Q23) that are *complex combinations* (comparisons of two aggregations). The limitation to simple aggregations comes from the limited expressivity of existing systems. As a consequence, all questions can be answered with Sparklis-Analytics, and its "Examples" page provides the solutions to the 50 test questions as "Open" links.

Methodology. We participated to the challenge by going through the 150 questions, building a table query in Sparklis-Analytics for each question, and submitting the found answers for the 50 test questions. In order to estimate the usability of our approach on real and large data, we also asked a non-IT person to go through the same process. Her education is in business studies, she works mostly with Excel, and had never used our tool before the experiment. We therefore submitted two set of answers: *expert* and *beginner*. We wanted to involve more non-IT subjects but going through that experiment was quite time-consuming (10h in total for the beginner

user). Still we believe that because of the very different profiles of those users our experiment is instructive from a qualitative point of view, and shows promising results. At the least, it demonstrates the possibility for a non-IT user to answer a large range of questions on real and complex data. To the best of our knowledge, no other tool achieves a similar result. During the test phase, we measured the clock time, including: question reading and understanding by the user, user actions in the tool, and computations by the tool and the endpoint.

Results and interpretation. Table 4 compares the performances of the expert and beginner users. The performance of the expert shows that virtually all questions (94%, $F_1 = 0.95$) can be answered accurately and efficiently (half questions answered in less than 1'30"), once the user is fluent in the use of the tool. Although the median time of the beginner is nearly three times higher (4') than for the expert, the success rate of 76% ($F_1 = 0.82$) is very satisfying for a beginner given the complexity of the data, and the novelty of the tool compared to Excel. Also, the maximum time per question does not differ drastically between the two users. The clock time measures of the expert imply that our tool is responsive because each question requires about 10 steps, each of which involves user action and SPARQL evaluation in addition to the computations specific to our approach. To compare with other challenge participants, which used automated question answering approaches, their success rate is 50% for QA^3 ($F_1 = 0.53$), and 38% for CubeQA ($F_1 = 0.44$).

The difference in success rate between *expert* and *beginner* is mitigated when looking at the cause of errors. Among the 6 beginner errors, 1 is explained by a lack of attention because the missed action was performed in several other questions, 3 are explained by a lack of exposure to a similar case during the training phase ("When" questions, numeric filters like "more than 10000000"), and 2 are explained by the difficulty to find the right property and value in a datacube. The 2 expert errors come from ambiguity in questions Q35 and Q42, which admit several equally plausible answers (e.g., several URIs have the same label). The explanation for unanswered questions is generally that the datacube is not explicit in the question. This requires to catch a first entity by string matching (e.g., "Research into Infrastructure" in Q33), which appeared to be inefficient and unreliable on the SPARQL endpoint. During the training phase, the expert user found a few errors in the manually crafted golden standard, which were reported to the organizers. At the QALD-6 workshop during ESWC'16, we were invited to give a talk in order to present our approach and results to the challenge (see [8] for detailed results).

7.3 Data Analytics at Persée

Persée[4] is a French organization that provides free access to more than 600,000 scientific publications, notably in the domain of humanities and social sciences. It maintains a SPARQL endpoint that gives access to their metadata[5], and they had already adopted Sparklis as an exploration and querying tool. We have contacted them in order to evaluate their need for analytical questions, in order to measure the adequacy of our approach to their needs. Without explaining them our approach or the range of questions that we cover, we collected a set of spontaneous questions they were interested in. To our surprise, those questions cover all kinds of uses cases presented in Section 2. We had rather expected simpler queries like retrieval queries, e.g. *all publications of some author after some date*. We here list a sample of representative questions, and then comment on them:

- Q1 How many documents a given person (e.g., Pierre Bourdieu) has co-authored with each of his co-authors?
- Q2 For each author, get the number and the list of the titles of his documents.
- Q3 How does the average number of authors per document evolves through time?
- Q4 Are there persons whose death date is before the birth date, thus revealing errors in data?
- Q5 Sort authors by decreasing duration of their activity period, computed as the difference between the maximal and the minimal publication date of their documents.
- Q6 For each publication year, get the number of articles published that year, and whose title matches a given term (e.g., "rural"), in order to study the evolution of that term across time.

Q1 and Q2 are aggregations, with two aggregators in Q2 ("the list of" translates to `GROUP_CONCAT`). Q3 is a nested aggregation, first counting authors per document, then averaging that count per publication year. Q4 involves a filter whose expression combines two properties of persons (birth date and death date). Q5 and Q6 are complex combinations of aggregations and bindings. In Q5, first two aggregations, a maximum and a minimum, then a binding for the difference, and finally a sorting. In Q6, first a filtering, then an aggregation. We did not receive any question that could not be answered with Sparklis-Analytics. This validates the relevance of our use cases, and the expressivity range of our table queries.

Note that the Persée's dataset is not at all organized into data cubes, and does not even contain numerical data, apart from dates. The central property is the one relating documents to authors, which is an *n-n* relationship. As the above questions show, both documents and authors can be used either as a dimension or as a measure (counting authors per document or counting documents per author). This real use case therefore validates our direct approach to RDF data analytics.

8 CONCLUSION AND FUTURE WORK

We have shown how the expressivity of SPARQL 1.1 can be leveraged to offer rich data analytics on vanilla RDF graphs. This includes OLAP-like analytical queries, and goes beyond with nested aggregations, and combinations with bindings and filters. We have also shown how to make it accessible to people without knowledge of SPARQL, through a NLI. We have implemented our approach, and validated its expressivity, usability, and scalability in real settings. Future work includes the addition of higher-level constructs (e.g., `'the proportion of'`) to simplify frequent complex combinations of computations; and the graphical visualization of results (e.g., charts, maps, timelines).

[4] `http://www.persee.fr/`
[5] `http://data.persee.fr/`

Acknowledgement. I wish to thank QALD organizers for the datacube task, Pierre-Antoine Champin and Cécile Almonté for valuable feedback and suggestions, and Eléonore Jouffe and the Miage students at Univ. Rennes 1 for their participation to the user studies.

REFERENCES

[1] M. Arenas, B. Cuenca Grau, E. Kharlamov, Š. Marciuška, and D. Zheleznyakov. 2016. Faceted search over RDF-based knowledge graphs. *Web Semantics: Science, Services and Agents on the World Wide Web* 37 (2016), 55–74.

[2] M. Atzori, G. Mazzeo, and C. Zaniolo. 2016. QA^3: a natural language approach to statistical question answering. (2016). http://www.semantic-web-journal.net/system/files/swj1847.pdf submitted to the Semantic Web journal.

[3] S. Chaudhuri and U. Dayal. 1997. An overview of data warehousing and OLAP technology. *ACM Sigmod record* 26, 1 (1997), 65–74.

[4] D. Colazzo, F. Goasdoué, I. Manolescu, and A. Roatiş. 2014. RDF analytics: lenses over semantic graphs. In *Int. Conf. World Wide Web*. ACM, 467–478.

[5] R. Cyganiak, D. Reynolds, and J. Tennison. 2013. The RDF data cube vocabulary. *W3C Recommendation (January 2014)* (2013).

[6] G. Demartini, I. Enchev, M. Wylot, J. Gapany, and P. Cudré-Mauroux. 2011. BowlognaBench – Benchmarking RDF Analytics. In *Data-Driven Process Discovery and Analysis*. Springer, 82–102.

[7] S. Ferré. 2016. Bridging the Gap Between Formal Languages and Natural Languages with Zippers. In *Extended Semantic Web Conf. (ESWC)*, H. Sack et al. (Eds.). Springer, 269–284.

[8] S. Ferré. 2016. SPARKLIS on QALD-6 Statistical Questions. In *Semantic Web Evaluation Challenge*. Springer, 178–187.

[9] S. Ferré. 2017. Sparklis: An Expressive Query Builder for SPARQL Endpoints with Guidance in Natural Language. *Semantic Web: Interoperability, Usability, Applicability* 8, 3 (2017), 405–418.

[10] P. Hoefler, M. Granitzer, V. Sabol, and S. Lindstaedt. 2013. Linked Data Query Wizard: A Tabular Interface for the Semantic Web. In *The Semantic Web: ESWC 2013 Satellite Events*. Springer, 173–177.

[11] K. Höffner, J. Lehmann, and R. Usbeck. 2016. CubeQA - Question Answering on RDF Data Cubes. In *Int. Semantic Web Conf.* Springer, 325–340.

[12] M. Kaminski, E. V. Kostylev, and B. Cuenca Grau. 2016. Semantics and expressive power of subqueries and aggregates in SPARQL 1.1. In *Int. Conf. World Wide Web*. ACM, 227–238.

[13] B. Kämpgen and A. Harth. 2011. Transforming statistical linked data for use in OLAP systems. In *Int. Conf. Semantic systems*. ACM, 33–40.

[14] V. Lopez, V. S. Uren, M. Sabou, and E. Motta. 2011. Is Question Answering fit for the Semantic Web?: A survey. *Semantic Web* 2, 2 (2011), 125–155.

[15] W. May. 1999. *Information Extraction and Integration with Florid: The Mondial Case Study.* Technical Report 131. Universität Freiburg, Institut für Informatik. Available from http://dbis.informatik.uni-goettingen.de/Mondial.

[16] B. Neumayr, C.G. Schuetz, and M. Schrefl. 2015. Towards Ontology-Driven RDF Analytics. In *Advances in Conceptual Modeling*. Springer, 210–219.

[17] L. Rietveld and R. Hoekstra. 2013. YASGUI: Not just another SPARQL client. In *The Semantic Web: ESWC 2013 Satellite Events*. Springer, 78–86.

[18] E. Sherkhonov, B. Cuenca Grau, E. Kharlamov, and E. V. Kostylev. 2017. Semantic Faceted Search with Aggregation and Recursion. In *Int. Semantic Web Conf. (ISWC) (LNCS 10587)*, C. d'Amato et al. (Eds.). Springer, 594–610.

[19] SPARQL11 2012. SPARQL 1.1 Query Language. (2012). http://www.w3.org/TR/sparql11-query/ W3C Recommendation.

[20] C. Unger, A.-C. N. Ngomo, and E. Cabrio. 2016. 6th Open Challenge on Question Answering over Linked Data (QALD-6). In *Semantic Web Evaluation Challenge*, H. Sack et al. (Eds.). Springer, 171–177.